\chapter{Future Work and Conclusions}\label{sec:conclusions}

\section{Future Work}\label{sec:conclusions:future\_work}

\subsection{Platform Independence} \label{sec:conclusions:future\_work:platform\_independence}

The prototype toolkit was implemented using development tools from Texas Instruments that are designed specifically for their TMS320 line of DSPs. The decision for using these tools was based on the assumption that they would simplify development, at the expense of platform independence. The key technology used that is not available for other platforms is the Real Time Operating System (RTOS) DSP/BIOS. DSP/BIOS was initially used for its debugging capabilities, but the end product only uses it for its multithreading capabilities.

\subsubsection{DSP/BIOS in the Prototype Toolkit}\label{sec:conclusions:future\_work:platform\_independence:dsp\_bios}

The threading capabilities of DSP/BIOS were both a gift and a curse. It makes a certain amount of sense to use threading with the firmware because multiple parts to the firmware are acting independently, however DSP/BIOS was a poor choice because its threading capabilities were heavily problematic. DSP/BIOS runs off of one of the reserved timers on the chip. When the timer interrupts, the DSP/BIOS task scheduler runs and determines if a context switch is required. The problem with this is that DSP/BIOS also uses that clock time to run all of its other functions, many of which cannot be fully disabled. If the timer is configured too fast, then DSP/BIOS consumes most of the processing time, preventing the user code from getting to run often enough. To fix this, the timer must be slowed so that DSP/BIOS does not consume too many processing resources, but this also has unpleasant side-effects if one tries to perform time-slicing (i.e. switching between multiple threads in succession give the appearance as if they are all running at the same time).

DSP/BIOS is not a pre-emptive multitasking operating system, meaning that time-slicing is not performed by the OS automatically, although their documentation claims, oddly enough, that it is pre-emptive without this feature (technically DSP/BIOS is a cooperative multitasking OS). In order to implement time-slicing in DSP/BIOS, one must explicitly call \lstinline$TSK\_yield$ to allow another thread the opportunity to run. Experimentation found that calling \lstinline$TSK\_yield$ incurs a one clock tick penalty. This means that the slower the clock tick, the longer the processor will be stuck doing nothing, which is in direct contrast to the processor resource utilization problem. Further experimentation found that a 1 ms clock tick results in roughly 10\% processor utilization by DSP/BIOS, but a 100 \textmu s clock tick results in nearly 95\% processor utilization, which is clearly unacceptable. However, a 1 ms clock tick results in a 1 ms delay between task switches, which is also unacceptable.

The solution employed for the toolkit in its current form was to eliminate all time-slicing, and instead manage everything through priorities. Fortunately, when a task with a higher priority than the currently running task is awakened by the running task, an immediate task switch occurs without a clock tick by the DSP/BIOS timer. This behavior allows the code to work properly through the use of carefully managed priorities. The application layer runs at the lowest priority, followed by the protocol stack, followed by the transmission side of the data-link layer, (implicitly) followed by the data-link layer interrupts. This method works, but did force the removal of some features from the MPI layer (notably \lstinline$MPI\_Isend$). In hindsight, DSP/BIOS was a poor choice for an RTOS due to these performance reasons. Even if performance wasn't an issue, DSP/BIOS would still be a poor choice due to its platform dependent nature.

\subsubsection{FreeRTOS} \label{sec:conclusions:future\_work:platform\_independence:freertos}

One promising RTOS replacement is FreeRTOS, an open source RTOS whose primary goal is to be lightweight. FreeRTOS officially supports 23 platforms, including ARM, TI MSP430, Atmel AVR, Freescale HC12 and Coldfire, and PIC processors. Some of these processors, such as the MSP430 and HC12 series, are low-end embedded products, which is a testament to the lightweight nature of the RTOS. To achieve this level of performance, FreeRTOS provides a minimum of features necessary to get multitasking working. FreeRTOS provides pre-emptive, cooperative, and hybrid configurations depending on the architecture. Most of the project is written in C, which makes porting to a new architecture relatively simple. FreeRTOS also provides execution trace capabilities and stack overflow detection, which is helpful for debugging. These features are precisely the features desired for this project. \cite{ref:2010-freertos-freertos}.

\subsection{Sockets API}\label{sec:conclusions:future\_work:sockets\_api}

Currently, MPI is the only application level API that can be used by application developers. Lower level functions exist that are fairly well encapsulated, but using these can be a bit messy from an API perspective. A future area of improvement would be to implement a subset of the BSD Sockets API. The Sockets API was first created in the early 1980's for the Berkley Software Distribution (BSD) Operating System, a UNIX-like operating system and has become the de-facto standard in sockets. Other networking APIs are heavily based on the BSD Sockets API. BSD Sockets provides support for both connection oriented (TCP) and connectionless (UDP) connections. This API would be especially useful when performing distributed I/O, where MPI may not be the best fit. \cite{ref:2003-stevens-unix\_networking\_programming}

\subsection{SequenceL} \label{sec:conclusions:future\_work:sequencel}

Functional programming is a different paradigm for programming, compared with imperative programming (i.e. procedural and object oriented). Functional programming languages are designed to express algorithms in a mathematical manner. In contrast, imperative programming languages are designed to express algorithms in the manner that they are executed on the target machine. Functional languages are based on lambda calculus, which enables formal reasoning about programs written in them. Common functional programming languages include Haskell, Scheme, and LISP. Functional languages are divided into two groups: pure functional languages and impure functional languages. Pure functional languages are not mutable, meaning that they do not contain mechanisms for expressing ``state.'' In short, assignment and I/O do not exist in a purely functional language. Impure function languages are mutable. Functional languages are ``referentially transparent,'' that is any object can be replaced by any other equivalent object. The ability to perform mathematical substitution is incredibly powerful when it comes to writing formal proofs about a program. Purely functional languages do not allow side-effects. Finally, functional languages allow higher order functions, meaning that functions can take functions as arguments and can return functions. \cite{ref:1996-goldberg-functional\_programming\_languages}

\subsubsection{Background}\label{sec:conclusions:future\_work:sequencel:background}

SequenceL is a relatively new purely functional programming language that is very compact and has some interesting qualities that make it a natural fit for parallel computing.

The only data structure available in SequenceL is the sequence, which can be used to represent a vast array of data structures. In addition to the standard mathematical operations found in most programming languages, two key operations appear in SequenceL: Consume-Simplify-Produce (CSP) and Normalize-Transpose (NT). A SequenceL program contains an initial ``tableau,'' a special sequence that contains the initial function calls (if any) and initial data. This tableau is simplified according to the rules of the basic operators, CSP, and NT until no more operations can be performed. The final tableau is the output of the program. \cite{ref:2005-cooke-sequencel\_overview}

CSP is the mechanism used to simplify an expression. The first step, consume, takes an expression from the tableau. The next step, simplify, attempts to simplify the expression using the rules available to it. Simplifications range from replacing a function call with the function definition, applying an NT, or performing a basic mathematical operation. The last step, produce, puts the simplified expression back into the tableau. \cite{ref:2005-cooke-sequencel\_overview}

NT is a mechanism that allows operations to apply to data that is of a different order. For example, the plus (+) operator is only defined for adding two scalars. The NT mechanism allows the plus operator to be applied to vectors without the need to redefine the plus operator. If CSP were to encounter the expression \lstinline$[1,3,5]\*2 $, then an NT is first applied. The first step in NT is normalize, which takes the  operator and the scalar  and normalizes them across the expression to form \lstinline$[1,3,5][\*,\*,\*][2,2,2] $. The next step in NT is to transpose, with in this case distributes the \lstinline$\*$ operator and the scalar \lstinline$2$ to form \lstinline$[1\*2,3\*2,5\*2] $, which now only consists of scalar multiplication. \cite{ref:2005-cooke-sequencel\_overview}

Looking back at the previous example, an opportunity for parallel computation exists. The three multiplication expressions in the final tableau can all be executed concurrently. It turns out that even the NT can be applied in parallel. This holds true for all ``atomic'' expressions because SequenceL does not allow side-effects or state. Another advantage of using a functional programming language is that there are never any dependencies, because writing is not allowed. Algorithms then become easy to parallelize automatically without the need for the programmer to identify the parallelisms.

\subsubsection{Challenges for Embedded Systems}\label{sec:conclusions:future\_work:sequencel:embedded\_challenges}

Several SequenceL interpreters and compilers currently exist, including parallel compilers, that target traditional workstations. However, some barriers to adoption exist for use in embedded systems. The primary barrier is that the compilers make extensive use of dynamic memory. Dynamic memory is problematic on processors without a Memory Management Unit (MMU), which most mainstream embedded processors lack. Without an MMU, memory fragmentation becomes a problem quickly, and most compilers forgo support entirely, or at most provide limited functionality. In order for SequenceL to be adopted in the embedded space, the compiler must first be re-written to not make use of dynamic memory, which is no small task. The primary concern with eliminating dynamic memory from the compiler is that sequences are allowed to grow during the execution of the program. The rest of this section is devoted to describing a mechanism to avoid dynamic memory. The other issues with porting the SequenceL compiler to embedded systems involve issues such as removing calls to libraries that don't exist in embedded systems (such as the Standard Template Library (STL)), which is not difficult.

\subsubsection{Embedded Variant Proposal}\label{sec:conclusions:future\_work:sequencel:embedded\_variant}

Due to the immutability of SequenceL, I/O must be handled via other mechanisms. Mixing SequenceL and C makes sense, then, because a) I/O handling for embedded systems is already handled well through C and b) SequenceL to C cross compilers already exist. SequenceL would be used to develop the processing algorithms, and C would be used to develop the I/O algorithms. The C program would be the ``primary'' language, with C programs making calls to SequenceL functions. Thus, the embedded SequenceL mechanisms need only to focus on function definitions, not complete tableaus.

The key to enabling a dynamic memory free compiler is to determine how much space is needed for a given sequence at compile time. Sequences can be of any dimension, and SequenceL does not require the dimension of a sequence to be defined programmatically. In addition, SequenceL does not provide mechanisms for defining the length of a given dimension. Taken together, determining memory usage for a given program without extra information not provided by the function definition is impossible. One possible means to provide this information would be to create a modified version of the SequenceL syntax and semantics that is specifically intended for embedded systems. This is undesirable because programs written in ``standard'' SequenceL would not be compatible with programs written in ``embedded'' SequenceL. The other approach is to use an ``interactive'' compiler that asks the users questions to gain this information. This approach has the advantage of not modifying the basic SequenceL language, thus making programs compatible and making the switch between embedded and non-embedded programming simpler for developers.

The primary question a compiler will ask the user is to define the initial dimension(s) and length(s) of function parameters. This does limit the flexibility of function calls, but given the nature of most embedded programs (i.e. little to no user interaction which necessitates these decisions at design time), the limitations are reasonable. With this knowledge, the compiler can run a specially modified SequenceL interpreter to determine the maximum memory size of all sequences in the program. Scenarios in which the maximum memory cannot be determined with this information alone exist. These scenarios are easiest to explain with an example:

\lstinline$Foo(n)::=1 when n=1 else n++Foo(n-1)$

This example creates the sequence \lstinline${n,n-1,...,2,1}$ using the concatenation operator (++) and recursion. The length of the sequence is determined by the \emph{value} of \lstinline$n$, not just its dimension and length. The second question that should be asked, then, is if the value(s) of any of the parameters are known. With this information, it is possible for most programs to be analyzed for their memory usage, however not all programs will resolve to known values.

The proposed method for determining maximum memory usage is a modified interpreter that performs an ``algebraic'' execution of the algorithm. The algebraic interpreter, as it will henceforth be called, replaces the parameters of the function with ``symbolic sequences.'' Take for example the following SequenceL implementation of matrix multiply:

\begin{lstlisting}

Prod(a,b)::=a\*b

\end{lstlisting}

The initial tableau in the interpreter would look like:

\begin{lstlisting}

{Prod(a,b)}

\end{lstlisting}

and, with user supplied dimensions of 1 and lengths of 4, would be modified to look like:

\begin{lstlisting}

{Prod({a1,a2,a3,a4},{b1,b2,b3,b4})}

\end{lstlisting}

where $\textrm{a}\_\textrm{x}$ and $\textrm{b}\_\textrm{x}$are atomic scalars. The execution would look like:

\begin{lstlisting}

{Prod({a1,a2,a3,a4},{b1,b2,b3,b4})}

=>{{a1,a2,a3,a4},\*,{b1,b2,b3,b4}}

=>{{a1,a2,a3,a4},{ \*,\*,\*,\*},{b1,b2,b3,b4}}

=>{{a1\*b1,a2\*b2,a3\*b3,a4\*b4}}

\end{lstlisting}

This execution shows us that the output will require a sequence of dimension 1 and length 4. An example execution of the recursion example shown earlier with the value for n specific would look like:

\begin{lstlisting}

{Foo(n)}

{Foo(3)}

{1 when 3=1 else 3++Foo(2)}

{3++Foo(2)}

{3++(2 when 2=1 else 2++Foo(1))}

{3++(2++Foo(1))}

{3++(2++(1 when 1=1 else 1++Foo(1)))}

{3++(2++(1))}

{3,2,1}

\end{lstlisting}

In this case, a final value was presented without any algebraic values because everything was known in advance. As a result, an algebraic interpreter execution can solve part of, if not all of, the problem. Incidentally, this information could be used for optimization purposes. Without the value of \lstinline$n$, in this example, calculating the maximum memory used would not be possible. When a problem is encountered, such as when \lstinline$n$ is undefined in this example, then the algebraic interpreter can inform the programmer where the problem is, so that it can be fixed.

\subsection{Potential Applications of the Toolkit}\label{sec:conclusions:future\_work:toolkit\_applications}

There are many applications that could benefit from a toolkit such as this one. Most algorithms have to make a tradeoff between processing time and memory consumption. Fast algorithms can be achieved at the expense of large memory requirements, and low memory usage algorithms can be achieved at the expense of processing time. Embedded systems tend to be both processing power and memory constrained, which makes it impossible for these types of algorithms to be implemented at all. Using this toolkit to parallelize an algorithm can get around these limitations. Many, if not most, parallel algorithms do not require the entire dataset to be present on all nodes, allowing the data to be spread around the system. Distributing the data around the system allows the use of data sets that cannot fit in the memory of a single system, in effect giving a memory increase in addition to the processing power increase.

Media compression schemes rely on mathematical transformations to change the data into forms that are more easily compressed. Common transformations used in media codecs are the Discrete Fourier Transform (DFT), the Discrete Cosine Transform (DCT), and the Discrete Wavelet Transform (DWT). The DWT has many advantages over the other two transforms and compression schemes based on the DWT, such as JPEG2000 \cite{ref:2001-taubman-jpeg2000} and BCWT \cite{ref:2006-guo-bcwt}, offer some of the best compression available. DWT based image codecs have been unable to find any traction in the market, however, in part because their high processing \emph{and} memory resource requirements prevented them from being implemented on digital cameras. This toolkit could be used to implement a fast DWT in embedded systems, such as higher-end digital cameras.

One aspect of the toolkit that has not been discussed so far is distributed computing that is *not* used for parallel computing. One potential application of this toolkit that is quite enticing for embedded systems is distributed I/O. Multiple embedded processors can be connected together using the toolkit to take advantage of the aggregate I/O. Say a system is needed that contains 2000 GPIO inputs. This cannot be done using a single processor (or even an FPGA for that matter), but using the toolkit, a couple of GPIO heavy processors can be coupled together to reach the 2000 mark.

\section{Conclusions} \label{sec:conclusions:conclusions}

\subsection{Purpose of the Toolkit} \label{sec:conclusions:conclusions:toolkit}

The toolkit serves as a useful, real-world test system for evaluating the routing algorithm. Due to a lack of existing solutions for embedded parallel and distributed computing, a new toolkit was implemented that provided features necessary for implementing the routing algorithm. Previous work on R\"acke's method has been purely theoretical, which is of limited value if it is never implemented or used elsewhere. By implementing a modified version of R\"acke's method in a real system, the usefulness of the method is more fully realized. A side-effect of this project is that the toolkit has also shown that parallel and distributed computing is possible with mainstream embedded processors and that application-specific interconnect mechanisms are not necessary.

The physical and data link layer have provided a good ``in-between'' mechanism that is faster than other mainstream mechanisms, such as $\textrm{I}^2\textrm{C} $ and CANbus, but is not as fast as high-end mechanisms, such as RapidIO and Ethernet. SPI by itself is rarely used for more than a couple of nodes due to the lack of a protocol and the necessity to configure each device as a master or slave, which this mechanism remedies. Having choice in interconnection technologies is important because different applications have different needs, and this mechanisms fills a current void in the interconnection market.

The network layer combined with the physical and data link layers forms a complete protocol for use in larger scale networks than is currently possible with $\textrm{I}^2\textrm{C} $ and CANbus. The protocol is not as complete as RapidIO, but then again RapidIO is not available on mainstream hardware. This protocol shows ``full-scale'' protocols for connecting multiple mainstream embedded processors is possible, something which has not been done before. Previous attempts at connecting mainstream embedded processors together for the purpose of parallel and distributed computing, such as in \cite{ref:2007-szekacs-multiprocessor\_spi\_system} and \cite{ref:2007-raman-h.264\_parallel\_transcoder}, did not produce results that could be applied to other distributed networks, and did not produce great results in general.

MPI is well established as \emph{the} method for programming distributed systems. This implementation shows that MPI, or at least a subset thereof, is suitable for embedded systems. This allows programmers to use their existing skills in MPI to program this system.

\subsection{Significance of the Routing Algorithm}\label{sec:conclusions:conclusions:significance:routing}

The author's contribution in the area of routing is a working non-deterministic oblivious routing scheme suitable for embedded systems that is based on Harald R\"acke's method. This implementation is, to the best of the author's knowledge, the first implementation of R\"acke's method in an actual system of any sort, embedded or otherwise. Oblivious routing algorithms are not currently well established in the computing world, but are the subject of ongoing research in the academic community because of the algorithm's low congestion. R\"acke's method guarantees that the maximum congestion is at most $log^3 n $ greater than a centralized offline algorithm, where everything is known in advance. \cite{ref:2003-racke-oblivious\_routing} This relationship is very significant in minimizing congestion in networks, and is especially important in minimizing congestion in embedded networks. Congestion implies that more resources are consumed, which are scarce in embedded systems. One downside to R\"acke's method is that all nodes in the network must be known when the routing tree is generated. Another downside to R\"acke's method is that calculating the routing tree and mapping a tree path to a network path is complex. Properly solving the algorithm requires the use of Linear Programming techniques that are NP-complete. Approximations exist that are solvable, but they are still far too complex for embedded systems. The first disadvantage actually makes the algorithm a good fit for embedded systems, because the nodes available will always be known due to the small size of embedded networks. The second disadvantage is a major hurtle for embedded systems.

Chapter \ref{sec:routing} presented a modified version of R\"acke's method that introduces new mechanisms for partitioning a graph and for selecting a network path. These new mechanisms make R\"acke's method very efficient, at the expense of guaranteed bounds on congestion. The algorithm leverages the strengths of oblivious routing while being efficient enough for embedded systems.

The method presented here, and oblivious routing in general, is not perfect and is considered in \cite{ref:2006-mavronicolas-eight\_open\_problems\_in\_distributed\_domputing} to be one of the eight open areas of research in distributed computing. The author of this article correctly points out that oblivious routing tends to favor minimizing congestion, sometimes at the expense of ``stretch,'' which is the ratio of the minimum path length and a given path length. The author suggests that attempting to optimize congestion and stretch simultaneously could produce an ideal routing algorithm. As it is, oblivious routing is still a very enticing routing mechanism for embedded systems.